Hinweise zur Implementierung der lexikalischen und semantischen Analyse

1 Flex, ein Scanner-Generator unter Unix

In diesem Abschnitt beschreiben wir die wesentlichen Eigenschaften des Scanner-Generators *flex*, der unter Unix als Public Domain Software zur Verfügung steht.

1.1 Die Arbeitsweise von Flex-generierten Scannern

Der Kern des Generators ist die Erzeugung eines deterministischen endlichen Automaten zur Erkennung einer regulären Sprache. Der erzeugte DEA erkennt maximal lange Präfixe der restlichen Eingabe als Symbole (token). Es gibt keinen Sieber, wie er in der Vorlesung beschrieben worden ist. Statt dessen gibt es eine Schnittstelle zu C, so daß benutzerdefinierte C–Funktionen bei Erreichen jedes Endzustandes aufgerufen werden können. Diese sind für die Weiterverarbeitung von gefundenen Symbolen zuständig. Das zuletzt gefundene Symbol steht über einen globalen Zeiger (`yytext`) zur Verfügung, seine Länge in einer globalen integer–Variablen (`yyleng`). Der ganze erzeugte Scanner ist eine C–Funktion `yylex`, welche von anderen Programmen, etwa einem Parser aufgerufen werden kann.

1.2 Flex–Spezifikationen

Die zu liefernde Flex–Eingabe gliedert sich in drei Abschnitte, getrennt jeweils durch '%%':

Definitionen
%%
Regeln
%%
C–Programme des Benutzers

Zur Vereinfachung der Beschreibung der Regeln führt der **Definitionen**–Abschnitt Namen für reguläre Ausdrücke ein. Diese Namen können, eingeschlossen in geschweifte Klammern, in späteren Definitionen und in Regeln anstelle dieser regulären Ausdrücke stehen.

Beispiel:

```
ZI [0--9]
BU [a - z]
ID {BU} ({BU} | {ZI})*
```

Die **Regeln** bestehen jeweils aus einem *Muster*, einem regulären Ausdruck, und einer *Aktion*, einem Stück C–Code. Dieser C–Code wird ausgeführt, wenn der erzeugte Scanner ein Symbol als zu diesem Muster gehörend erkennt.

Beispiel:

```
{ZI}+     printf ("Eine ganze Zahl: %s (%d)\n", yytext, atoi(yytext));
"+"|"-"|"*"|"/"  {
          printf( Ein Operator: %s\n", yytext);
          }
if|then|else|while|do|begin|end {
          printf ("Ein Schluesselwort: %s\n", yytext);
          }
{ID}      printf ("Ein Name: %s\n", yytext);
[ \t\n]   /* konsumiert Leerzeichen, Tabs und Zeilenenden */
```

Einige Bemerkungen zu diesem Beispiel:

- Zwischen eckigen Klammern, '[' und ']', kann man Zeichenklassen aufführen, wie im Definitionsteil auch durch Angabe von Bereichen. Einzelne Zeichen und Zeichenklassen können auch negiert auftreten.
- '*', '+', '—' als Metazeichen haben die übliche Bedeutung. Treten sie als Zeichen der spezifizierten Eingabe auf, werden sie durch Doppelapostroph geschützt.
- Die Schlüsselworte `if`, ..., `end` sind natürlich auch Mitglieder der Klasse `ID`. Der Konflikt zwischen den Endzuständen zu diesen regulären Ausdrücken wird zugunsten der Schlüsselwörter entschieden, da sie früher in der Spezifikation stehen.

Die **C–Programme des Benutzers** umfassen im wesentlichen die in den Aktionen der Regeln angewandten Prozeduren. Ihr Auftreten innerhalb der Spezifikation ist jedoch nicht zwingend. Ein Includieren der entsprechenden Datei, in der sie definiert sind, erzielt den gleichen Effekt.

2 Bison, ein LALR(1)–Parsergenerator

Die zweite Phase des Frontends repräsentiert die syntaktische Analyse. Ihre Aufgaben bestehen im wesentlichen aus der Analyse der syntaktischen Programmstruktur, der zugehörigen Behandlung fehlerbehafteter Eingaben und der Berechnung der Zwischendarstellung, auf der die nachfolgenden Compiler–Phasen arbeiten. Diese Phasen erweitern die Zwischendarstellung um bestimmte Attribute (z.B. Typ–, Adreßinformationen), so daß für deren Werte Platz in der Knotenstruktur bereitgestellt werden muß. Insbesondere ist im Rahmen dieses Praktikums darauf zu achten, daß die Zwischendarstellung den vom bereits implementierten Backend erwarteten Aufbau besitzt.

Analog zum lexikalischen Analysator ist auch bei den Aufgaben eines Scanners eine Standardisierung festzustellen. Diese hat zur Entwicklung von Parser–Generatoren geführt. Zur Implementierung des Scanners verwenden wir daher den Generator *bison*. Im folgenden geben wir eine kurze Einführung in die Arbeitsweise von *bison* und das Format seiner Benutzer–Eingaben, so daß die Studie des in der Vorlesung ausgegebenen Skripts erleichtert wird.

2.1 Die Arbeitsweise von Bison–generierten Parsern

Bison ist ein LALR(1)–Parsergenerator, der ein Teil des GNU–Systems der Open Software Foundation repräsentiert. Er ist aufwärts kompatibel mit dem LALR(1)–Parsergenerator Yacc, einem Unix–Dienstprogramm. Wie
Yacc erlaubt auch Bison die Eingabe mehrdeutiger Sprachgrammatiken und die Beseitigung der daraus resultierenden Parser–Konflikte, insbesondere bei Operatoren, durch die Angabe von Präzedenzen und Assoziativitäten. Außerdem bietet Bison die Erzeugung eines rudimentären Mechanismus zur Behandlung von Syntaxfehlern an.

Wesentliche Benutzereingabe für den Parsergenerator ist die Grammatik der zu analysierenden Programmiersprache. Es steht eine Schnittstelle zu C zur Verfügung, die die Angabe benutzerdefinierter Fuktionen erlaubt. Mit der Reduktion einer Grammatikregel wird dann der Aufruf der entsprechenden Benutzerfunktion verbunden. Diese kann z.B. den Aufbau des Teilbaums der Zwischendarstellung bewirken, der der angesprochenen Grammatikproduktion entspricht. Weiterhin gibt es zu jedem Grammatiksymbol genau ein abgeleitetes Attribut. Sein Vorkommen auf einer linken Produktionsseite wird mit '$$', sein Vorkommen beim i–ten Symbol der rechten Seite mit '$i' bezeichnet. Standardtyp dieses Attributs bei allen Symbolen ist *int*, jedoch sind abweichende Typvereinbarungen möglich.

Von Bison erzeugte Parser werden mit *yyparse*() aufgerufen. Sie erwarten die Existenz eines Scanners namens *yylex*. Dieser kann z.B. von *flex* erzeugt oder auch handgeschrieben sein.

2.2 Bison–Spezifikationen

Eingabespezifikationen für Bison besitzen das folgende Format:

%{
C Deklarationen
%}
Bison Deklarationen
%%
Grammatik–Regeln
%%
Benutzerdefinierter C–Code

Der *C–Deklarationsteil* beinhaltet Typ– und Variablendeklarationen, Makrodefinitionen und erforderliche *include*'s, die in den folgenden Teilen der Eingabe benötigt und an den Anfang des generierten Parsercodes gestellt werden.

Der *Bison–Deklarationsteil* listet die Terminal– und Nichtterminalalphabete auf und ordnet dabei den Attributen der jeweiligen Grammatiksymbole Typen zu, falls diese vom Standardtyp *int* abweichen (siehe Beispiel 2). Weiterhin können hier die Präzedenzen und Assoziativitäten von Operatoren angegeben werden (siehe Beispiel 1).

Beispiel 1:

```
%token NUM     /* Symbolklasse ganze Zahlen */
%left '-' '+'  /* linksassoziative Operatoren */
%left '*' '/"
%left NEG     /* unaeres Minus */
%right '^'    /* rechtsassoziativer Potenz-Operator */
```

Die Auflistung der Operatoren bestimmt die Präzedenzen; Operatoren in der gleichen Zeile haben die gleiche Präzedenz, später eingeführte höhere als früher eingeführte. NEG wird als Präzedenzstufe eingeführt, um das unäre Minus einzuordnen. Dieses kann erst aufgrund der syntaktischen Position vom binären Minus unterschieden werden.

Es folgt ein Beispiel für die Verwendung verschiedener Attributtypen. Nehmen wir an, daß ein abstrakter Syntaxbaum gesteuert durch die Syntaxanalyse aufgebaut werden soll. Dann könnte z.B. der Knoten zu einer Anweisung Verweise auf seine Komponentenbäume haben, der Knoten zu einem Bezeichner einen Verweis in die Symboltabelle und der Knoten zu einer Konstanten einen Verweis auf eine Darstellung der Konstanten. Die Attribute zu den entsprechenden Nichtterminalen wären dann Zeiger auf `struct`s mit den erforderlichen Komponenten. Durch eine *union*–Deklaration im Bison–Deklarationsteil wird die Menge der möglichen Attributtypen eingeführt.

Beispiel 2:

```
%union{
   symrec *symbol_ptr;
   snode  *node_ptr;
   value  *value_ptr:
}
```

Den Attributen der Nichtterminale wird anschließend jeweils einer der so eingeführten Attributtypen zugeordnet.

```
%type <symbol_ptr> identifier
%type <node_ptr> statement, ifstat, whilestat
%type <value_ptr> const
```

Die *Grammatikregeln* geben die Produktionen und die mit ihnen assoziierten semantischen Aktionen an.

Beispiel 3: Eine Grammatik für variablenfreie, arithmetische Ausdrücke mit semantischen Regeln zu ihrer Auswertung.

```
expr      NUM               { $$ = $1;      }
        | exp '+' exp       { $$ = $1 + $3; }
        | exp '-' exp       { $$ = $1 - $3; }
        | exp '*" exp       { $$ = $1 * $3; }
        | exp '/' exp       { $$ = $1 / $3; }
        | '-' exp %prec NEG { $$ = - $2;    }
        | exp '^' exp       { $$ = pow ( $1, $3 );}
        | '(' exp ')'       { $$ = $2;      }
;
```

Die Mehrdeutigkeiten dieser Grammatik werden durch die Assoziativitätsregeln und die Präzedenzen aus Beispiel 1 aufgelöst.

Der *benutzerdefinierte C–Code* kann die Definitionen der Aktionen, die in Grammatikregeln auftreten, oder auch die Definition eines Scanners `yylex` und von weiteren benötigten Funktionen enthalten.

2.3 Fehlerbehandlung

Findet ein Bison–erzeugter Parser einen syntaktischen Fehler, so ruft er eine Routine `yyerror` auf, die eine Fehlermeldung ausgibt und normalerweise die Syntaxanalyse beendet.

Bison unterstützt jedoch wie Yacc die folgende einfache Art der Behandlung von Syntaxfehlern. Es gibt ein vordefiniertes Symbol `error`. Dieses Symbol kann an beliebiger Stelle auf rechten Seiten von Produktionen stehen. Der Bison–erzeugte Parser meldet ein Vorkommen dieses Symbols, wenn er keinen legalen Übergang hat. Enthält der aktuelle Zustand ein Item, in dem `error` auf den Punkt folgt, so wird ein Übergang unter `error` ausgeführt. Folgt eine Reduktion mithilfe dieser Produktion, so kann mit der zugehörigen Aktion eine Fehlerbehandlung vorgenomen werden. Die vordefinierte Routine `yyerrok` veranlaßt den Parser zur Fortsetzung der Analyse.